Parallelization

Aryan Philip, Eugene Kim, Gabrielle Rackner, Jiaying Yang,
Kevin Benavente, Kira Fleischer, Paulami Bhattacharya,
Peiyuan Zhang, Runlong Su, Sharanya Prabhu,
Stanley Woo, Tianyu Fan, Yiwen Tu, Paulami Bhattacharya

February 2025

1 Post-Training Quantization (slides 1-16)

As a review of quantization from the previous lecture, its purpose is to reduce the precision of numerical weights and activations to make models more efficient in terms of memory and computation. The following table summarizes the storage and compute requirements of two quantization methods discussed in the previous lecture:

	No quantization	**K-Means-based Quantization**	**Linear Quantization**
Storage	Floating point weights	Integer weights with a floating-point codebook	Integer weights
Compute	Floating point arithmetic	Floating point arithmetic	Integer arithmetic

1.1 Quantization Granularity

There are three main quantization granularities that affect how weights and activations are quantized: per-tensor, per-channel, and group. Each method provides different trade-offs in terms of computational efficiency, memory savings, and model accuracy.

1.1.1 Per-Tensor Quantization

Per-tensor quantization is a technique in which a single scale (S) and zero point (Z) are used to quantize an entire tensor, such as all the weights of a layer or all the activations in a layer. This is the simplest and most computationally efficient approach, as it requires fewer scaling factors and ensures the same precision for all weights. However, it may lead to a loss of accuracy, especially if the tensor contains large outlier weights or the range varies wildly across channels. Figure 1 shows an example of the wide range of weights possible in

a tensor, which can lead to less accurate quantizations. A better approach is per-channel quantization.

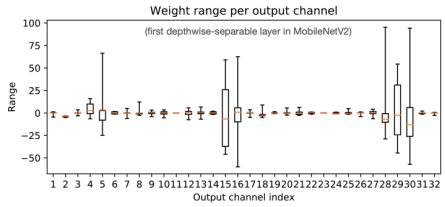


Figure 1: Wide range of weights in a tensor

Example 2-bit symmetric linear quantization using the per-tensor method. We will lose precision for small numbers if we only using a single scale S for the whole weight tensor.

2.09	-0.98	1.48	0.09
0.05	-0.14	-1.08	2.12
-0.91	1.92	0	-1.03
1.87	0	1.53	1.49

Binary	**Decimal**
01	1
00	0
11	-1
10	-2

1. Find $|r|_{max} = 2.12$
2. Calculate $S = \frac{|r|_{max}}{q_{max}} = \frac{2.12}{1} = 2.12$
3. Using $Z = 0$ (since this is symmetric quantization), compute the quantized weights as $q_W = \text{round}(\frac{r}{S} + Z) = \text{round}(\frac{r}{2.12})$:

1	0	1	0
0	0	-1	1
0	1	0	0
1	0	1	1

4. Reconstruct the weights by multiplying the quantized weights by the scale (compute $q_W S = 2.12 q_W$)

2.12	0	2.12	0
0	0	-2.12	2.12
0	2.12	0	0
2.12	0	2.12	2.12

5. Compute the quantization error $||W - S \odot q_W|| = 2.28$

1.1.2 Per-Channel Quantization

Per-channel quantization is where a separate scale (S) and zero point (Z) are assigned for each channel in a tensor, such as each output channel of a convolutional layer. This allows for better preservation of numerical precision since different channels may have very different ranges, as seen above in Figure 1. Although this method reduces quantization error by providing more fine-grained quantizations, it is slightly more computationally expensive than per-tensor quantization since you are computing more scales (S) and storing more.

Example 2-bit symmetric linear quantization using the per-channel method.

2.09	-0.98	1.48	0.09
0.05	-0.14	-1.08	2.12
-0.91	1.92	0	-1.03
1.87	0	1.53	1.49

Binary	**Decimal**
01	1
00	0
11	-1
10	-2

1. Find $|r|_{max}$ for each row (channel):

$$r_{max} = 2.09$$
$$r_{max} = 2.12$$
$$r_{max} = 1.92$$
$$r_{max} = 1.87$$

2. Calculate $S = \frac{|r|_{max}}{q_{max}}$ for each row (channel):

$$S = 2.09$$
$$S = 2.12$$
$$S = 1.92$$
$$S = 1.87$$

3. Compute the quantized weights per row (channel) as $q_W = \text{round}(\frac{r}{S})$:

1	0	1	0
0	0	-1	1
0	1	0	-1
1	0	1	1

4. Reconstruct the weights by multiplying the quantized weights by the scale for that row (compute $q_W S$ per row/channel):

2.09	0	2.09	0
0	0	-2.12	2.12
0	1.92	0	-1.92
1.87	0	1.87	1.87

5. Compute the quantization error $||W - S \odot q_W|| = 2.08$

1.1.3 Group Quantization

The most fine-grained quantization method is group quantization where weights are divided into groups of elements and each group gets its own scale and zero point, such as quantizing per vector. This is the most accurate method with lowest quantization error because you are essentially providing no quantization, but that is also a drawback of this method since you are essentially doing no quantization.

1.2 Multi-Level Quantization

Combining granularities of quantization can create the ideal results for a particular algorithm. Multi-level quantization applies quantization at various levels of granularity such as tensor and vector quantization. Scale values are applied at each granularity. The quantization may scale by group as seen in the equation below (with gamma). The equation below represents two-level quantization. In the second equation, S_q is the scale factor for each vector and γ is the scale vector for each tensor.

$$r = S(q - z) -> r = \gamma S_q(q - Z)$$

Multi-level quantization can expand beyond two-level even though that is normally the "sweet spot" for most deep learning algorithms. As we continue to increase the amount of group levels, the computation cost increases. Below is the updated equation for n number of groups.

$$r = S(q - z) -> r = S_{l_0} S_{l_1} ... S_{l_n}(q - Z)$$

1.3 Linear Quantization on Activations

Applying linear quantization on activation weights looks slightly different than applying quantization on the numerical weights. Quantized weights are static and activation values range across inputs. In order to calculate the quantized activations, activation statistics are calculated *before* the model is deployed and used to determine the clipping range (r_{min}, r_{max}).

$$\hat{r}^{(t)}_{max,min} = \alpha r^{(t)}_{max,min} + (1 - \alpha)\hat{r}^{(t-1)}_{max,min}$$

The clipping range is typically a dynamic range for activations which is determined using a moving average. The range is averaged across multiple

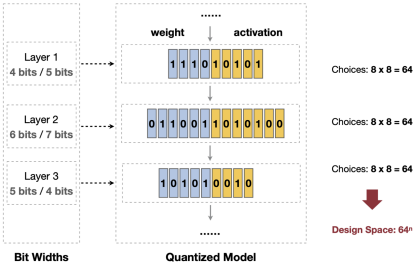


Figure 2: An illustration of the exponentially exploding search space for a mixed-precision training deign.

training steps. To avoid including outliers in the clipping range, one can utilize calibration of the initial weights. If outliers are included in the dynamic range, the clipping range is not representative of the activation distribution.

2 Mix-Precision Training

2.1 Introduction (Slides 17-21)

In standard quantization, all layers typically use the same bit-width (e.g., uniform 8-bit quantization). However, this approach does not account for the varying robustness of different layers to precision reduction. Therefore, instead of uniform quantization, we can apply non-uniform, mixed-precision quantization, where more robust layers are quantized more aggressively to lower bit-widths, preserving precision where needed.

Large Design Space One main challenge facing mixed-precision training is its large design space, as shown in Figure 2. With many layers in deep networks, manually determining the optimal bit allocation is labor-intensive. For example, as illustrated in Figure 4, given an 8-bit budget per layer for weights and activations respectively, the total search space grows exponentially as $(8 \times 8)^n = 64^n$ for a n-layer model, rendering exhaustive search infeasible.

Automatic Design One solution would be automating the mixed-precision quantization design using machine learning techniques. More concretely, we can develop a cost model to predict the trade-offs between bit-width selection and accuracy/latency, and use algorithms to efficiently explore mixed-precision configurations, balancing performance and efficiency. This approach enables efficient mixed-precision quantization, significantly improving both accuracy and latency compared to uniform quantization, without the need for exhaustive manual tuning.

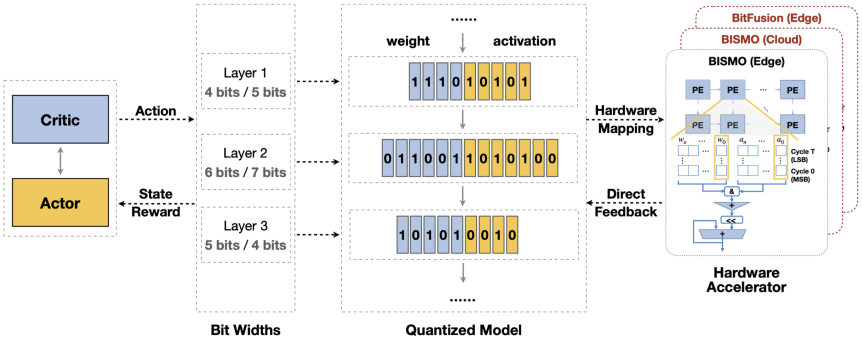


Figure 3: An illustration of mixed-precision training with automated search in the tremendous design search space.

2.2 Mixed Precision Training (Slides 22 - 24)

Many Neural Networks some layers are naturally more sensitive to dynamic range than others. For example, Normalization. If we use a lower bit precision, we will lose a lot of accuracy. The intuition is, we need to be very precise hence we need to allocate more bits for fractions. Similarly, SoftMax requires us to do calculations with an exponential function. Again, we must add dynamic range by allocating more bits on both exponent and fraction. Another consideration is when your accumulating the network gradients, we are faced with very similar issues earlier mentioned. Based on a scheme developed by NVIDIA, we are suggested to use full precision (FP32) for those sensitive operations. After those operations are complete we can essential downcast precision to half precision for more robust operations resulting in more memory efficient operations. This scheme can be formalized into a standardized 16-32 mix-precision pipeline illustrated by Figure 4.

2.3 Memory Usage in Mix-Precision Training (Slide 25-26)

For training a 175B-parameter GPT-3 model, the memory requirements are calculated as follows:

- **Parameters**: 175×10^9 parameters $\times$ 2 bytes/fp16 = 350 GB
- **Activations**: 96 (batch size) $\times\, 3.2 \times 10^6$ (tokens) $\times$ 12,288 (dimensions) $\times$ 2 bytes = 7,488 GB

Memory usage for optimizer states:

- **Master copy (fp32)**: 4×175 B = 700 GB

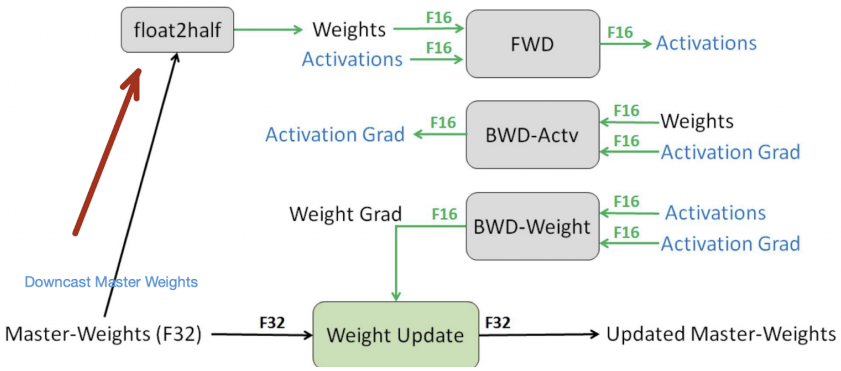


Figure 4: An illustration of a standardized 16-32 mix-precision pipeline.

- **Gradients (fp16)**: 2×175 B $= 350$ GB
- **Running copy (fp16)**: 2×175 B $= 350$ GB
- **Adam momentum states**: $2 \times 4 \times 175$ B $= 1{,}400$ GB

The theoretical lower bound for memory consumption in mix-precision training is:

$$(4 + 2 + 2 + 4 + 4) \times N = 16N \text{ bytes, where } N = \text{number of parameters.}$$

2.4 Scaling Down Machine Learning (Slide 27)

Deploying machine learning on edge devices remains a critical objective and active research area. While scaling up ML infrastructure is dominated by a few major industry players, the scaling down market faces significant fragmentation. This diversity stems from the vast array of edge device architectures and use cases. Such fragmentation presents opportunities for startups to address niche challenges, but poses challenges for large enterprises seeking standardized solutions across heterogeneous hardware platforms.

3 Overview of Parallelization (Slides 28–29)

The discussion on parallelism concludes the comprehensive overview of ML systems. This section begins by examining the rationale for parallelization through an analysis of technological trends. It then systematically explores key concepts including: fundamental principles of ML parallelism, essential collective communication patterns, data parallelism strategies, model parallelism approaches (encompassing both inter-operator and intra-operator variations), and emerging auto-parallelization techniques.

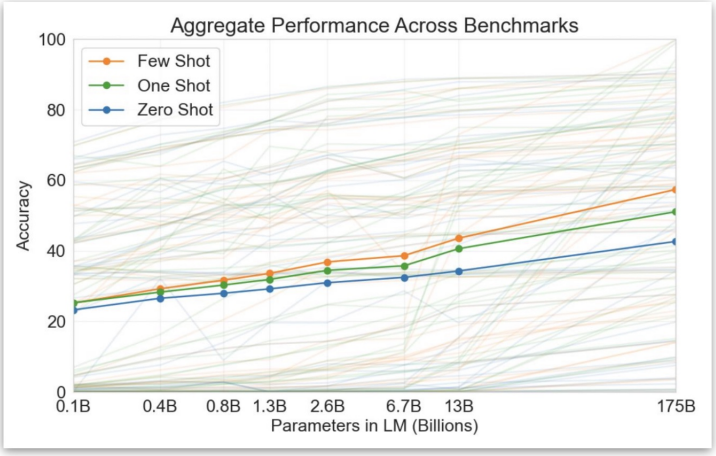
4 Parallelization (Slides 30-40)

4.1 End of Moore's Law

According to Moore's Law, the number of transistors in a dense integrated circuit doubles approximately every two years, driving exponential growth in CPU and GPU capabilities. However, this fundamental principle that guided technological advancement for decades is now facing multiple challenges in the era of artificial intelligence.
While Moore's Law remained valid for many years, it no longer adequately describes the current pace of AI model growth, as model sizes are increasing by a factor of 10 every 18 months. For instance, the DeepSeek model has reached an impressive scale of 600 billion parameters, illustrating the extraordinary rate at which model sizes are expanding.

4.2 Why do we develop large models?

The drive toward larger models is primarily motivated by two key factors:

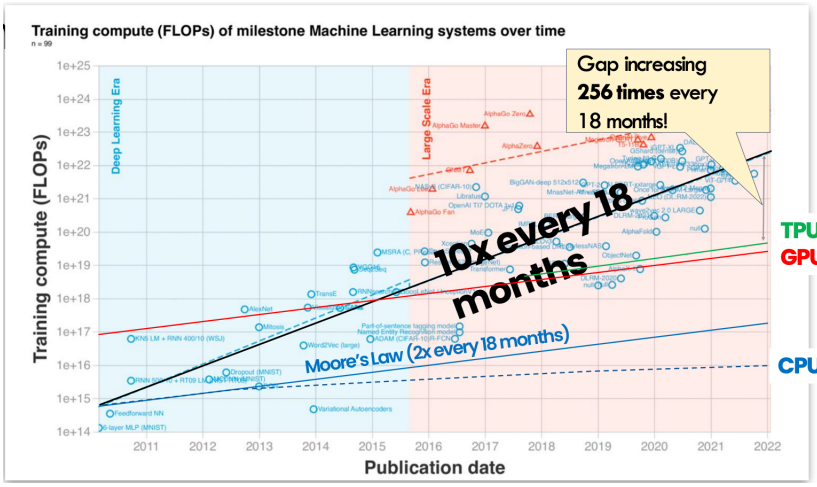


1. **Enhanced Performance and Accuracy**
   Empirical evidence demonstrates that scaling models leads to improved performance across multiple benchmarks. As seen in the graph research has shown consistent improvements in model capabilities across 20 different benchmarks as model size increases. Some researchers hypothesize that achieving near-perfect accuracy across a wide range of tasks could be a stepping stone toward Artificial General Intelligence (AGI).

2. **Emergent Capabilities**
   Larger models exhibit emergent capabilities - newfound abilities that weren't explicitly programmed but arise from the scale of the system. Modern large language models demonstrate reasoning capabilities, including the

ability to solve complex mathematical problems, understand nuanced context, and perform multiple tasks without specific training for each one.

4.3 The Growing Challenge



4.3.1 Computational perspective

This rapid scaling of AI models is creating an unprecedented demand for computational resources, particularly high-performance chips. The gap between model growth and hardware capabilities is widening, as GPU and CPU advancement, still broadly following Moore's Law, cannot match the exponential growth in model sizes. This divergence presents a significant challenge for the future of AI development and raises important questions about sustainable scaling practices.

If we project the growth of accelerator capabilities into the future, a substantial gap remains between the computational power required and what is currently achievable with CPUs. The floating-point operations per second (FLOPs) of GPUs continue to lag behind demand, while TPUs offer only a marginal improvement over GPUs. Despite these advancements, the computational gap is widening exponentially, increasing by a factor of 256 every 18 months. Using a single accelerator is insufficient to keep pace with the increasing model sizes necessary for development. From a computational standpoint, the required processing power is growing at an unsustainable rate.

4.3.2 Memory perspective

From a memory perspective, the situation is even more extreme—the memory needed to store and train models increases by a factor of 35 every two years. In contrast, GPU memory growth follows a much flatter trajectory, making it impossible to bridge this gap with current hardware trends.

Historically, the largest model that could fit within the memory of a single GPU was BERT. Since then, model sizes have expanded to the point where storing parameters alone now requires more than 100 GPUs.

5 Types of parallelism (slides 41-51)

This data highlights the necessity of parallelization strategies, specifically **data parallelism** and **model parallelism**.

5.1 Data and model parallelism

There are two types of Parallelism, namely, Data and Model Parallelism. Data parallelism is relatively straightforward. At the accelerator level, it follows the principles of Single Instruction, Multiple Data (SIMD). In GPUs, this means distributing data across numerous cores, with each core handling a portion of the computation. Similarly, across multiple GPUs, each device processes a subset of the data. In this lecture, we dive deeper into scenarios involving multiple GPUs wherein each GPU is given a part of the data. Each GPU is installed on some node and is given a part of a huge dataset.

A core assumption of data parallelism is that the whole model fits on a single GPU. If the model does not fit on a single machine, we must adopt model parallelism. The model is basically a computational graph. It involves many 'MATMUL' and non-linear functions. How we partition a model is equivalent to thinking about how we partition tensor operators. Intuitively, the first technique is to cut in the middle by feeding the first few layers of the model to a single GPU and the next remaining layers to another GPU. Further, we can pipeline the execution to make it more efficient. The second technique is to cut horizontally instead of vertically. This becomes rather complicated because each layer has different operators, and each NN can have different kinds of layers (Conv, Embedding), different types of clusters (e.g., 8 GPUs in one node, big cluster with thousands of GPUs distributed in different ways).

5.2 Computational and system perspective

The classical view of ML parallelism is to classify parallelism into data and model parallelism. We will now map this intuition into our definition of models, data, and compute to approach it from a computational and system perspective. Recapping the master equation from the earlier lecture, from the computational perspective, we care about compute and memory. In the equation, for compute, the elements we require are firstly the delta, which performs forward and backward computation. We parallelize by partitioning the compute function across different GPUs. Secondly, we require the optimizer f, where we optimize gradients to parameters. We apply function f across different GPUS. This is the main compute in the equation. On the memory side, we have two primary

sources of memory consumption. D, the data or the number of batches we need to partition D, and θ, wherein we partition the parameters.

5.3 The communication challenge

Communication is crucial. In a single device, communication happens locally, across the memory hierarchy, and is easy. However, here, with distributed GPUs, it is challenging to communicate across the interconnect of devices. Interconnects like Nvidia's NVLink (GPUs in one box), networks (Amazon elastic fabric, TCP/IP) are complex. Communication is much slower in these cases, with high latency. Handling communication parameters and activations depends on the NN graph partitions.

5.4 Parallelism workflow

At a high level, for data parallelism, each GPU is given different data, and they compute on their own. At some point, synchronization is needed to make progress. We will need to communicate parameters and gradients. In data parallelism, the parameters and model code are replicated across the GPUs, and they perform the respective computation on the specifically designated batch of data. They will produce copies of gradients. These gradients need to be synchronized. All GPUs' gradients have to be accumulated and applied to parameters. This generates a new copy of parameters, which then need to be redistributed across the GPUs.

$$\theta^{(t+1)} = f\left(\theta^{(t)}, \nabla_L\left(\theta^{(t)}, D^{(t)}\right)\right)$$

where, θ represents the parameter, f represents the weight update function (like SGD, Adam, etc.), $nabla_l$ refers to the model (like GPT, CNN, etc.), and D refers to the data at time t.

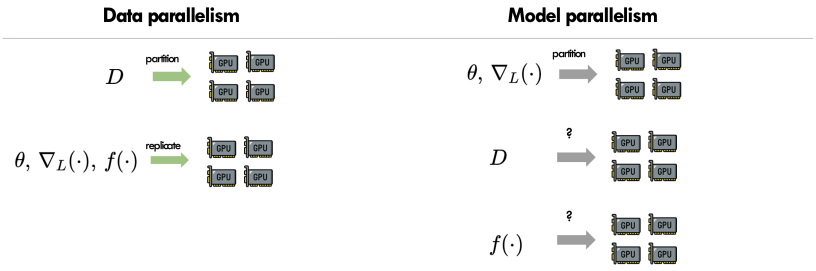


Figure 5: Parameter distribution for data and model parallelism.

In model parallelism, we partition the model. Either partition the model parameters (θ) or partition the gradient (∇_l) computation function. We need to figure out how to deal with data (partition or not partition). Further, we need to partition the optimizer states. Many options to partition are subject to

network latency and many such constraints. We need to aim to partition in a smart way.

5.5 Parallelism in action

To bring more clarity, we shall revisit the three parts of our model and view them as computational graphs, namely, forward, backward, and weight update graphs. We shall read them as a cluster. For example, in the Nvidia DGX V100 cluster, they connect linearly with 8 GPUs, with each GPU with a multiway NVLink. NVLink has high bandwidth, being very fast, almost like memory hierarchy. But, if one of these GPUs has to talk to another GPU outside this box, it requires ethernet. It requires movement from GPU memory to CPU, and CPU would send through ethernet. The bandwidth of ethernet is much slower than that of NVLink.

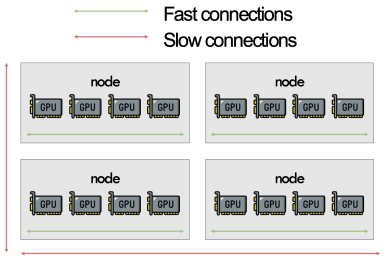


Figure 6: A typical GPU cluster topology

To make it more abstract, the right side figure represents a cluster with four nodes in which the green lines represent fast connections. GPUs communication between different boxes uses slow connections.

Hence, parallelization can be treated as partitioning of a computational graph on a device cluster by being subject to memory constraints and communication bandwidth. After partitioning, we want it to be as fast as possible without additional memory. This is the problem definition.

6 Intra and Inter parallelisms (Slides 52 - 62)

Parallelization = Partitioning Computation Graph on Device Cluster.
The problem that we have is : How to partition the computational graph on the device cluster? (Subjecting to memory constrain and communication bandwidth.)

6.1 Communication Characteristics

- **Inter-op parallelism**:

  - Assign different operators to different devices. The second op depends on the output of the first op.
  - Typically requires *point-to-point* communication between consecutive operators (e.g., sending outputs from Device 1 to Device 2).
  - Potential *device idle time* when an operator finishes early and must wait for other operators to complete their tasks before proceeding to the next stage.

- **Intra-op parallelism**:
  - Splits a single operator across multiple devices (e.g., large matrix multiplication).
  - Typically relies on *collective* communication (all-reduce, all-gather, broadcast, etc.) to merge partial results.
  - High throughput when well-implemented, but communication overhead can be significant if the operator is not large enough or if the network is slow.

6.2 Example

Consider we have the following computational graph, and we are given with 2 devices. We can follow the example in Figure 7

There are two strategies that we can think of:

- Strategy 1 - Inter-Operator Parallelism: Where device 1 takes the first matmul layer, while device 2 takes the remaining layers.
- Strategy 2 - Intra-operator Parallelism: In this case, we are cutting the operators, the weights, the input, and outputs half and half, sharing among two devices. (Both horizontally and vertically)

We can also do a horizontal cut, where device 1 takes the $w1$ and $w2$ and putting the rest on device 2.

Question: Is data parallelism intra or inter parallelism? Answer: It is intra-parallelism.

There are multiple intra-op strategies for a single node (Figure 8):

- Row-partitioning The input matrix is partitioned by rows, each partition (shown in blue/red) processes different rows of the input, operations like matmul are performed on these row partitions independently.
- Model-partitioning (Megatron) This is a hybrid of model and tensor parallelism with the workload is split across different devices. Each colored section (blue/red) represents different parts of the model running on different devices, Operations like matmul and relu are distributed across devices.

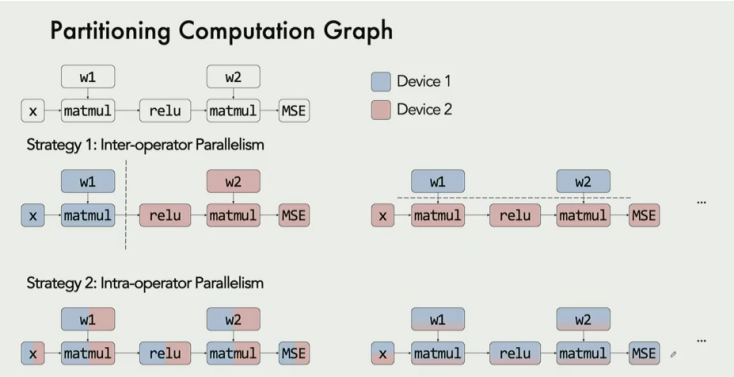


Figure 7: An illustration of how parallelism works

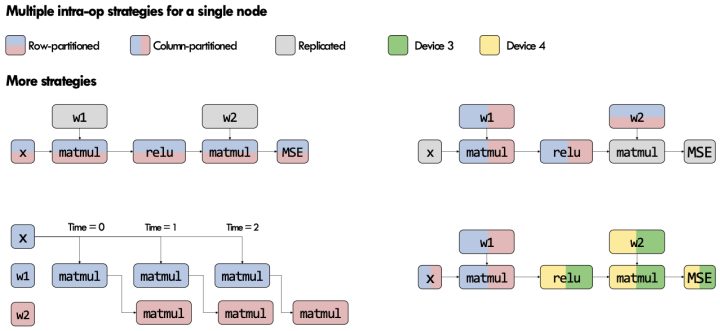


Figure 8: More Examples on Parallelisms

- Pipeline parallelism A way for temporal parallelism, operations are executed in a staged manner across time (Time = 0, 1, 2), different parts of the model (w1, w2) work on different timesteps, Enables better hardware utilization through pipelined execution.
- Multi-Device model partition Nowadays, models size is growing in a dramatic way hence multi-device model partition is necessary for training model in such size.

6.3 Trade-offs

- **Inter-op parallelism** can be simpler to implement if the operators are already relatively independent. However, one might encounter significant *idle times* or synchronization bottlenecks when operators are chained. The communication is called P2P and it is much less bandwidth compared with collective communication.

- **Intra-op parallelism** can deliver high speedups for computationally heavy operators (like large matrix multiplications), but collective communication steps can become bottlenecks. As we all know, collective communication operation like all-reduce is very costy, so the intra operation parallelise is better to be adapt to data cluster that gots higher innter bandwith.
- The optimal parallelism strategy often involves a *hybrid* approach, mixing inter-op and intra-op parallelism depending on the operator, the device topology, and the batch size.

7 Contributions

1. Kira Fleischer: Section 1
2. Gabrielle Rackner: Section 1
3. Yiwen Tu: Section 2.1
4. Eugene Kim, Kevin Benavente Section 2.2
5. Tianyu Fan: Section 2.3 - 3
6. Paulami Bhattacharya, Jiaying Yang: Section 4
7. Sharanya Prabhu, Aryan Philip: Section 5
8. Stanley Woo, Peiyuan Zhang, Runlong Su: Section 6